The design of the Boost interval arithmetic library

Hervé Brönnimann, Guillaume Melquiond, Sylvain Pion

The design of the Boost interval arithmetic library ⋆

Hervé Brönnimann [a,1] Guillaume Melquiond [b,2] Sylvain Pion [c,3]

[a] *CIS, Polytechnic University, Six Metrotech, Brooklyn, NY 11201, USA.* hbr@poly.edu

[b] *École Normale Supérieure de Lyon, 46 allée d'Italie, 69364 Lyon cedex 07, France.* guillaume.melquiond@ens-lyon.fr

[c] *INRIA, BP 93, 06902 Sophia Antipolis cedex, France.* Sylvain.Pion@inria.fr

Abstract

We present the design of the Boost interval arithmetic library, a C++ library designed to efficiently handle mathematical intervals in a generic way. Interval computations are an essential tool for reliable computing. Increasingly a number of mathematical proofs have relied on global optimization problems solved using branch-and-bound algorithms with interval computations; it is therefore extremely important to have a mathematically correct implementation of interval arithmetic. Various implementations exist with diverse semantics. Our design is unique in that it uses policies to specify three independent variable behaviors: rounding, checking, comparisons. As a result, with the proper policies, our interval library is able to emulate almost any of the specialized libraries available for interval arithmetic, without any loss of performance nor sacrificing the ease of use. This library is openly available at `www.boost.org`.

Key words: Interval arithmetic, software library, generic programming, policy-based design, robust computations, floating-point filter.

⋆ A preliminary version of this article appeared under the title "The Boost interval arithmetic library"at the 5th Conference on Real Numbers and Computers, 3-5 September 2003, Lyon, France.

[1] Work by the first author was supported by NSF CAREER Grant CCR-0133599.

[2] Work by the second author was partially accomplished during a visit to Polytechnic University, with support from ENS Lyon.

[3] Work by the third author was partially supported by NSF CAREER CCR-0133599 while visiting Polytechnic University, and conducted during a postdoc fellowship of NFS/ITR Grant CCR-0082056 at New York University.

1 Introduction

Interval computations, known as either *Interval Analysis* or *Interval Arithmetic* (both abbreviated as IA), are a way to extend the usual arithmetic on numbers to intervals on these numbers. The fundamental property which makes interval computations useful is the *inclusion property*: the extension $[f]$ to intervals of a function f is an interval $[f]([x_1], \ldots, [x_n])$ which must contain the image by f of every value $(x_1, \ldots, x_n)$ in the function domain. Thanks to this property, interval extensions can be used both for keeping track of roundoff errors as well as testing a property directly on a range of values. Ensuring this property requires access to the proper rounding modes of the arithmetic operations (most commonly, this can be controlled when using floating point computations that comply with the IEEE 754 Standard [1]).

The domains of applications of IA are wide and varied, ranging from data processing and integrating measurement errors, to geometric representation of solids and computer graphics, constrained global optimization, reliable computing (including integration, ODE and PDE solving, monitoring roundoff error propagation with a strong emphasis on linear algebra), and have found many applications (see [7,8,9] for an overview). Increasingly a number of mathematical proofs have relied on global optimization problems solved using branch-and-bound algorithms with interval computations (most famously, Hales' recent proof of Kepler's conjecture; [4] check also the proofs of optimality of various circles packings in squares and discs [5]). It is therefore extremely important to have a mathematically correct implementation of interval arithmetic.

There are many implementations of interval computations for different purposes (see section 3.2). Since the intended domains of applications sometimes have different semantics, or requirements, these implementations differ in the details. For instance, the meaning of comparisons may be different (section 3.1.3), or the behavior in exceptional situations may have different requirements (section 3.1.2). In some contexts like computer graphics, the mathematically correct inclusion property may not even be desirable due to lower speed, and an inclusion property within the roundoff errors may be acceptable. In others, special hardware support can be used to optimize the underlying rounded computations (section 3.1.1).

We present the design of the Boost interval library, which was accepted after a thorough review by the members of the Boost community. Its purpose is to provide a single C++ class template, `interval<T>`, and supporting functions,

[4] See http://www.math.pitt.edu/~thales/kepler98/ and refs. therein.
[5] See http://www.packomania.com/ for a listing of the known packings, with references to the literature.

whose behavior can be adapted to the various contexts mentioned above. The design goals were flexibility (the ability to modify the semantics of the basic interval type), ease-of-use, and at the same time without loss of efficiency (which is extremely important; for instance, branch-and-bound algorithms can typically run for several days). We settled for a policy-based design [2] and identified three orthogonal behaviors: *rounding*, *checking*, and *comparisons*. All three have several possible choices and we provide a few concrete policies for each. Almost all possible combinations are possible, and with a single design we can emulate a wide collection of interval types. We present this design in section 3.

In the conference version of this paper, we illustrated the basic usage of the library to compute the sign of a determinant using the algorithms of [4], and supplied many details that were not provided in their paper, including a few improvements that led to better computational efficiency or precision. This part did not provide new theorems or algorithms and was removed for reasons of space from the present article. The interested reader is referred to the conference version.

2 Background on interval arithmetic

Basic interval arithmetic is presented in many references (e.g. [6,12,17]). The purpose of this section is not to review the basic properties of interval arithmetic, but merely to set up the necessary notation and discuss those aspects which will be relevant to the design (next section).

Interval classification and bounds. Interval arithmetic deals with *intervals*: closed and convex sets of a totally ordered set $\mathbb{F}$, called the *base number type*. The base number type can be the set of real numbers $\mathbb{R}$, but this is not necessary. It could be a subset $\mathbb{F}$ of $\mathbb{R}$, such as algebraic numbers, rational numbers, floating-point numbers, even integers. Or it could also be a superset of those (Puiseux series, infinitesimal extensions, etc.).

There are only three kinds of closed and convex subset of a totally ordered set: the empty set, bounded sets, and unbounded sets. If $\mathbb{F}$ is unbounded in any direction, by introducing up to two new elements $-\infty$ and $+\infty$ respectively lesser and greater than any element in $\mathbb{F}$, each interval can be represented by a pair $[a, b] = \{x \in \mathbb{F} \mid a \leq x \leq b\}$.

When the property $a \leq b$ does not hold, the interval $[a, b]$ is the empty set. Some authors [17] specially handle the case of the unordered pair $[+\infty, -\infty]$ in order to represent a projective infinity. (For instance, it simplifies the interval version of the Newton algorithm.) The rules needed to handle such an infinity

are not naturally derived from the operations on $\mathbb{F}$, however, so this will not be considered here.

Consequently, we denote by $[x] = [\underline{x}, \overline{x}]$ an interval, $\underline{x}$ its *lower bound*, and $\overline{x}$ its *upper bound*. (Some authors use other representations, e.g. center-radius, which is less expressive than the above representation. We will not cover it here.)

Rounding and reliable computations. In computer arithmetic, we usually work with a subset of $\mathbb{F}$, called the *representable* numbers. Naturally, not all the intervals of $\mathbb{F}$ can be described if these numbers are used for the bounds. This subset may not even be closed with respect to the operations defined on the elements of $\mathbb{F}$.

Since we cannot always represent exactly the result of an operation on representable numbers, we must therefore compute in each arithmetic step the smallest interval $\diamond[x] = [\bigtriangledown\underline{x}, \bigtriangleup\overline{x}]$ that encloses $[x]$ such that $\bigtriangledown\underline{x}$ and $\bigtriangleup\overline{x}$ are representable, an operation called *rounding*. This means that $\bigtriangledown\underline{x}$ (respectively $\bigtriangleup\overline{x}$) is the next representable number to $\underline{x}$ (respectively $\overline{x}$) when rounding downwards (respectively upwards). If there is no such representable number, the corresponding infinity $-\infty$ or $+\infty$ should be given to ensure the inclusion property of interval arithmetic. [6]

First computing bounds in $\mathbb{F}$ and then rounding them to representable bounds is only an abstract construct, however. In an actual implementation, rounded operations must be used, such as in $[x] \oplus [y] = [\underline{x} \underline{+} \underline{y}, \overline{x} \overline{+} \overline{y}]$. [7]

Basic operations on bounded intervals. Operations and functions are extended to interval functions by the inclusion property: $f([x]) = [f(x)] = \{f(x) \mid x \in [x]\}$ for univariate functions, and similarly for multivariate by considering x a vector and $[x]$ a product of intervals. For instance, if both $[x] = [\underline{x}, \overline{x}]$, $[y] = [\underline{y}, \overline{y}]$ are bounded intervals, we set

$$
\begin{aligned}
[x] + [y] &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\
[x] - [y] &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}] \\
[x] \times [y] &= [\min\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}, \max\{\underline{x}\underline{y}, \underline{x}\overline{y}, \overline{x}\underline{y}, \overline{x}\overline{y}\}] \\
[x] \div [y] &= [x] \cdot [1/\overline{y}, 1/\underline{y}] \text{ if } 0 \notin [y].
\end{aligned}
$$

Similar definitions apply to other non-rational functions ($\sqrt{}$, sin, cos, etc.).

[6] If the implementation does not support infinities, a suitable behavior like throwing an exception should be adopted.

[7] Note that IEEE Standard 754 guarantees that $\underline{x} \underline{+} \underline{y}$ is the same as $\bigtriangledown(\underline{x} + \underline{y})$, but it suffices in general that $\underline{x} \underline{+} \underline{y} \leq \bigtriangledown(\underline{x} + \underline{y})$ to ensure the inclusion property.

Unbounded and empty intervals. The operations on unbounded intervals are deduced from the ones on bounded intervals. If the addition and the subtraction of $\mathbb{F}$ are naturally extended to handle infinities, then the previous definitions of addition and subtraction can be reused. Note that since an interval with $+\infty$ as a left bound or $-\infty$ as a right bound is empty, the undefined operation $\infty - \infty$ can never arise.

For the multiplication, the previous definition cannot be directly reused since the case $0 \times \infty$ can arise. To correctly handle it, the intervals must be classified depending on the sign of their elements. There are four possibilities:

- the zero interval $[0, 0]$,
- "positive" intervals ($\underline{a} \geq 0$ and $\overline{a} > 0$),
- "negative" intervals ($\overline{a} \leq 0$ and $\underline{a} < 0$),
- intervals crossing zero ($\underline{a} < 0$ and $\overline{a} > 0$).

The formulas defining the multiplication can now be simplified depending on the "sign" of the interval operands. For example, if $[x]$ and $[y]$ are both positive, the operation becomes: $[x] \times [y] = [\underline{xy}, \overline{xy}]$. These new definitions can now directly be used with infinite bounds. A detailed description and justification of such an interval arithmetic system can be found in [6] for example.

Finally, any operation involving an empty interval will return the empty interval. Indeed, the inclusion property mandates that $f(\emptyset) = \{f(x) \mid x \in \emptyset\} = \emptyset$.

Division and other functions. We have defined interval division by canonical set extension of the division in $\mathbb{F}$. However, the answer may not be an interval when dividing by an interval containing 0. For example, $1/[-1, 1]$ is the union of two intervals: $[-\infty, -1] \cup [1, +\infty]$. Because of unbounded intervals, there are even some cases where the result is not a finite union of intervals: $1/[1, +\infty] = [0, 1] \setminus \{0\}$.

Contrarily to [17] where signed zeros are used, this second case will not be handled specially: the smallest enclosing interval will be returned ($[0, 1]$ in the second example). However, for the first case, there is more than a single point that should be included in order to get the smallest enclosing interval. Consequently, two versions of the division can be defined: one of them will answer a single enclosing interval ($[-\infty, +\infty]$ in the first example) and the other will answer a union of intervals (at most 2).

This problem is not limited to the division. Any function which is only piecewise continuous on $\mathbb{R}$ can exhibit a similar behavior. For example, $\tan[\pi/4, 3\pi/4] = [-\infty, -1] \cup [1, +\infty]$. In the following, we will simply suppose that these functions return the smallest enclosing interval.

Comparisons. So far, all the operators previously described are restricted to the single set $\mathbb{F}$. However, more sets could be mixed. In particular, comparison operators can be defined since $\mathbb{F}$ is totally ordered and they are functions of $\mathbb{F} \times \mathbb{F} \to \mathbb{B}$ (where $\mathbb{B}$ is the Boolean set). Consequently, such an operator could be naturally extended to intervals and the result would be a Boolean interval.

There are four Boolean intervals: $\emptyset$, $\{\text{false}\}$, $\{\text{true}\}$ and $\{\text{false}, \text{true}\}$. There would be no problem if these were directly used by the application. The conditionals commonly used in numerical algorithms only have two branches, however, and never four. Naturally, $\{\text{false}\}$ could be mapped to false, and $\{\text{true}\}$ to true. The result is $\emptyset$ if and only if one operand is itself empty. But even when forbidding the two operands to be empty (an implementation could throw an exception in this case), there still is a third state $\{\text{false}, \text{true}\}$.

The inequality $[\underline{x}, \overline{x}] < [\underline{y}, \overline{y}]$ is $\{\text{false}, \text{true}\}$ when $\exists x_1, x_2 \in [\underline{x}, \overline{x}]\ \exists y_1, y_2 \in [\underline{y}, \overline{y}]\ x_1 \geq y_1$ and $x_2 < y_2$. There are at least three possible behaviors. The first one would be to forbid such a situation. The second one would be to always map this value to false. And the last one would be to always map this value to true.

The first solution behaves exactly like the operator on the base number type; an algorithm would not require any rewriting to switch from the base number type to an interval type. (Again, an implementation could simply throw an exception if the forbidden behavior occurred.) The second one implies a "certainly" semantics: the result is true if and only if $\forall x \in [\underline{x}, \overline{x}]\ \forall y \in [\underline{y}, \overline{y}]\ x < y$. And the third one means "possibly:" the result is true if and only if $\exists x \in [\underline{x}, \overline{x}]\ \exists y \in [\underline{y}, \overline{y}]\ x < y$.

Other orders on intervals can be useful in specific applicative contexts. For example, since intervals are subsets of $\mathbb{F}$, another natural order on them is the inclusion relation. This time, it is a Boolean function and it can be directly used in a program. However it has two drawbacks: it is only a partial order (and consequently may require a special handling of the incomparable state) and it is not related to the base order on $\mathbb{F}$. An example of total order is the lexicographic relation on various attributes of intervals, such as left bound, right bound, center, radius, width, and so on.

3 The design of the Boost interval library

The general principle of interval arithmetic described in the previous paragraph is applied to get a C++ class template `interval<T>`. There are many details to fill in, however, such as the representation of the empty interval,

what happens for exceptional values of the base types (such as floating point NaNs—*not a number*), how to perform the rounding (or not, if so desired). Typically these choices are made with a particular domain of application in mind.

The goal of the Boost interval library is to provide a *generic* implementation of interval arithmetic, in that those choices can be specified by the user of the library. For this, we use a mechanism called *policies* [2,16].

3.1 Overview of the policy-based design

The class template `interval<T>` actually has two template parameters, `T` and `Policies`, a default being provided for the second. The second parameter does not influence the data representation of an interval; it only describes the way the various algorithms will handle the data. It contains a reference to two types: the rounding policy and the checking policy. A policy is a class that specifies the behavior. For instance, the rounding policy will have operations for performing all the basic operations on the base type, with a specified rounding mode. [8]

There are two important consequences to using C++ templates as the mechanism for implementing policies as opposed to, say, an object-oriented framework. One is that C++ templates are implemented using static binding, and hence the policy is known at compile-time. This helps compilers to optimize. This is one of the most important aspects of the design.

The second is that the complete C++ interval type not only contains `T`, but also the policies. This is useful to prevent automatic casts from `interval<T,P1>` to `interval<T,P2>`, e.g., in case the checking policy of `P1` allows empty intervals but not `P2`.

Strictly speaking, the comparison mechanism is not a policy, since doing so would have incorporated the comparison into the interval type. We view comparisons not as intrinsically part of the interval type, as it is perfectly legal (and sometimes useful) to compare the same intervals in different ways (for instance, "certainly" comparisons for non-overlapping intervals, then some special treatment for overlapping ones).

[8] For example, $\underline{+}$ and $\overline{+}$ are such operations and the library functions rely on this policy to compute them.

3.1.1 The rounding policy

Since interval operations can be expressed in terms of rounded operations on the bounds, the library functions rely on a kernel of arithmetic functions to compute their results. The kernel to use is indicated by the rounding policy.

The library could have directly used a predetermined kernel for each base type. But it is not interesting to only have one kernel for a given base type, nor is it possible to predetermine a kernel for every user-defined base type. For example, suppose the class representation is an interval with rational bounds and the user wants to compute an enclosure for $\sqrt{2}$. Choosing among different policies, the user may prefer a rounding policy that computes small rationals for $\sqrt{2}$, or one that gives a better approximation but takes longer, etc.

These kernels are also responsible for all the optimizations specific to the type of bounds; in particular, when manipulating hardware floating-point numbers. Here is a common example of such an optimization. Processor floating-point units are usually unable to efficiently handle rounding mode changes: the latter breaks the execution flow and increases latency. In order to amortize the cost of these switches, a sequence of operations can be done with one fixed rounding mode. For example, if the current rounding mode is towards $+\infty$, the sum $a \underline{+} b$ can be computed as $-((-a)\overline{+}(-b))$ without changing the rounding mode. All this can be encapsulated within the arithmetic kernel. The functions on intervals do not need to know about these optimizations; they only ask the rounding policy to perform $a \underline{+} b$, no matter how.

Thanks to these two stages, the functions of the library are totally generic and can handle any types of intervals, yet the library is as fast as specialized libraries since the various kernels can be fully optimized for a particular base number type.

3.1.2 The checking policy

This policy allows the user to choose how the interval class will deal with exceptional cases. In particular, empty intervals may play an important role in some algorithms. However, there are also situations where they cannot occur, or where the user does not want them to be generated (a division by $[0, 0]$ should generate an empty interval but it can also mean there is a bug in the algorithm).

An interval function deals with empty intervals by asking the checking policy to verify if the input intervals are empty or not. If the user is sure no empty intervals will ever be created, the policy can disable these tests and the compiler will do a lot of dead code elimination.

It is generally safe to think that if no empty interval can be passed as input, no empty interval will be produced by a function, hence the checking can be done away with. But it can also happen that a function needs to output an empty interval (generally when the interval is outside the domain of the function, $\arccos[-32, -25]$ for example). In this case, the function will ask the checking policy to create such an interval. Depending on the choice of the policy by the user of the library, an empty interval will be created (letting the program proceed), or an exception will be thrown (an equally acceptable behavior since computing $\arccos[-32, -25]$ was probably a bug).

Empty intervals are not the only special case. The library functions should also be cautious and test each input for invalid data (a *Not a Number* of the IEEE 754 standard for example). But it is an overhead that is not always desired and that can be turned off by choosing the appropriate policy. For all these exceptional cases, the checking policy allows the user to select a particular behavior of the library.

3.1.3 Comparisons

The comparison scheme is made available through namespace resolution. The main advantage is that the policy is selected locally, so that several incompatible policies may be used in different portions of a same program. Moreover it allows to use the infix notation (operators `<`, `<=`, `>`, `>=`, `==`, `!=`) rather than calling some functions.

Thanks to this model, many comparison semantics can be defined. In particular, the library provides a lexicographic order (no real mathematical meaning but sometimes useful when manipulating data), an order based on the set inclusion partial order (subset, superset, proper subset, etc). There is also a namespace of operators that do not return a Boolean but a tristate value to reflect the previous mapping.

The default behavior of the operators is meant to reflect the results of the operators on the base numbers. So when the comparison is certain, the answer is `true`. When no pair of elements can satisfy the comparison, the answer is `false`. And when no definite answer can be given (generally because the intervals overlap), an exception is thrown.

3.2 Comparison with other C++ libraries

Many libraries and applications provide interval arithmetic. We compare with five of them that are typical implementations. Others can be found on the Interval web page [7].

Profil/BIAS [10], CGAL [5] and Gaol [14] only handle bounds of type `double`. Filib [11] and Sun [15] are a bit better in the sense that they are able to handle the three floating-point types `float`, `double`, and `long double` thanks to a template parameter. So these four libraries are restricted to hardware floating-point numbers.

MPFI [13] handles multi-precision floating-point numbers (as given by the MPFR library), so there is a bit of flexibility since the precision can be chosen. Again, however, only floating-point numbers are supported and none of these libraries are able to manipulate intervals with rational or integer bounds for example.

Profil/BIAS also suffers from always switching rounding modes. Thanks to BIAS level 1 and 2 techniques, the number of switches is reduced when computing with matrices and vectors. But not all programs manipulate matrices and they will then suffer from performance degradation. CGAL, Sun, Filib and Gaol libraries provide better rounding mode handling to avoid this overhead, but their use may change the floating-point semantic of the program.

Speed is not the only concern, validity of the results can also be a problem. For example, not all these libraries are able to correctly compute the product $[-1, 0] \times [5, +\infty]$ and the result will be absurd [9] (Profil/Bias will return $[-\infty, \mathrm{NaN}]$ for example, although the correct answer would have been $[-\infty, 0]$).

These libraries also have specific behavior in regard to empty intervals. They may test for them and handle them (MPFI) or throw an exception. They can also ignore them and produce invalid results. The behavior is fixed and sometimes not clearly defined; and moreover, the user cannot change it. Finally, none of these libraries allows to use specially crafted infix comparison operators and the user can only rely on functions.

By using the correct policies, the user should be able to emulate all the existing libraries with the Boost library, without sacrificing the precision, validity and speed of the computations. But, more important, the user is able to define a lot of new behaviors that were impossible to accomplish with these libraries.

[9] Having an interval with an infinite bound can easily result from an overflow when dealing with floating-point numbers. Consequently, even if an interval arithmetic library does not allow the user to explicitly create an unbounded interval, such an overflow can still happen and the library should be able to handle such intervals.

Given a polynomial P (represented by an array `P` and its size `sz`) and a value x, the following example computes the sign of $P(x)$. The answer is 1 if $P(x)$ is positive, -1 if $P(x)$ is negative, and 0 if the function does not compute the answer safely. We can start from a standard floating-point implementation. It uses Horner's scheme to compute the value of the polynomial at the given point and then evaluates the sign of this value. However the result will not be guaranteed because of roundoff errors. (See Figure 1, left.)

In order to get a guaranteed result, we just have to switch to interval computations. The only modifications are changing the type of the variable `y` and selecting the comparison scheme. (See Figure 1, middle.) As shown by this example, the necessary modifications to switch to interval computations are short and easy.

As it is, the code could be improved by informing the library that the user will only use mathematical functions involving IA and that the library is allowed to use whatever possible means to speed up the computations. It requires a new interval type to be defined since this information will be passed thanks to the interval policies. An object `rnd` (for *rounding*) is created and all the optimizations will take place during its lifetime. (See Figure 1, right.) If the processor supports hardware floating-point computations, the optimization will consist in setting the rounding mode upwards for the whole life of `rnd` and doing all the computations with this mode (using the opposite trick presented in section 3.1.1).

Such an optimization is essential for common processors like x86, Sparc, PowerPC and so on. Without it, a program that intensively uses interval arithmetic on hardware floating-point numbers would suffer from a huge slowdown. Indeed, switching the rounding mode of the floating-point unit usually requires to flush the entire processor pipeline. It is not unusual for a program to be slowed down by a factor ten in such conditions. It is the reason why the library offers an easy way to work around these limitations.

This maneuver is well documented and simple to set up. On the other hand, it cannot be made the default behavior, since it requires user cooperation. If it were, a careless or uninformed user may obtain erroneous results. It is why we have decided to leave it as a manually triggered optimization.

4 Conclusion

This paper introduces a new interval arithmetic library. It is a generic C++ library able to handle any kind of bounds (hardware floating-point numbers, rationals, multi-precision software numbers, etc.), and emulate a wide vari-

```
int
sign_polynomial(double x,
                double P[],
                int sz)
{
  // This version uses double
  // floating-point evaluation.
  // It may result in wrong sign.

  double y = P[sz-1];
  for(int i=sz-2; i >= 0; --i)
    y = y * x + P[i];

  if (y > 0.0) return 1;
  if (y < 0.0) return -1;
  return 0;
}
```

```
int
sign_polynomial(double x,
                double P[],
                int sz)
{
  // This version uses interval
  // evaluation.  Use default
  // rounding policy and choice
  // below of comparison policy.

  interval<double> y = P[sz-1];
  for(int i=sz-2; i >= 0; --i)
    y = y * x + P[i];

  using namespace
        compare::certain;
  if (y > 0.0) return 1;
  if (y < 0.0) return -1;
  return 0;
}
```

```
int
sign_polynomial(double x,
                double P[],
                int sz)
{
  // No unnecessary switching
  // of rounding mode
  typedef interval<double> I_aux;
  I_aux::traits_type::rounding rnd;
  typedef unprotect<I_aux>::type I;

  I y = P[sz-1];
  for(int i=sz-2; i >= 0; --i)
    y = y * x + P[i];

  using namespace
        compare::certain;
  if (y > 0.0) return 1;
  if (y < 0.0) return -1;
  return 0;
}
```

Fig. 1. Three versions of Horner's polynomial sign evaluation.

ety of behaviors (thus emulating various existing libraries). Performance has not been sacrificed to this genericity, however. In particular, with hardware floating-point numbers, on a complex algorithm (the determinant sign computation of [4] for example), the overhead of interval arithmetic can be made optimal, even when the algorithm involves intensive computations like matrix operations. An integration between Boost.interval and Boost.uBlas (a generic implementation of BLAS, the basic linear algebra subroutines) could provide the same level of optimization as Profil and BIAS, and is under consideration.

References

[1] ANSI/IEEE. *IEEE Standard 754 for Binary Floating-Point Arithmetic*. IEEE, New York, 1985

[2] A. Alexandrescu. *Modern C++ Design*. Addison Wesley, 2001.

[3] Boost. *The Boost C++ Libraries*. http://www.boost.org/

[4] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Disc. Appl. Maths* 109:25–47, 2001.

[5] CGAL. *Computational Geometry Algorithms Library*. http://www.cgal.org/

[6] T. Hickey and Q. Ju and M. H. Van Emden, Interval arithmetic: From principles to implementation. *J. ACM*, 48(4):1038–1068, 2001.

[7] *Interval Computations*. http://www.cs.utep.edu/interval-comp/

[8] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis, with Examples in Parameter and State Estimation, Robust Control and Robotics*. Springer-Verlag, 2001.

[9] R. Baker Kearfott and V. Kreinovich (eds.) Applications of Interval Computations. Kluwer, 1996.

[10] O. Knueppel. PROFIL/BIAS - A Fast Interval Library. *COMPUTING* Vol. 53, No. 3-4, p. 277-287. http://www.ti3.tu-harburg.de/knueppel/profil/

[11] M. Lerch, G. Tischler, J. Wolff von Gudenberg, W. Hofschuster, and W. Krämer. *FILIB++ Interval Library*. http://www.math.uni-wuppertal.de/org/WRST/software/filib.html

[12] R.E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.

[13] N. Revol and F. Rouillier. *MPFI 1.0, Multiple Precision Floating-Point Interval Library*. http://www.ens-lyon.fr/~nrevol/mpfi_toc.html

[14] *Gaol, Not Just Another Interval Library*. http://www.sourceforge.net/projects/gaol/

[15] Sun Microsystems. *C++ Interval Arithmetic Programming Reference*. http://docs.sun.com/db/doc/806-7998

[16] D. Vandevoorde and N. Josuttis. *C++ templates: the Complete Guide*. Addison Wesley, Boston, 2002.

[17] G. William Walster. *The Extended Real Interval System*. Manuscript, 1998. Preprint available at http://www.mscs.mu.edu/~globsol/walster-papers.html